Cursus OOP - Object-oriented programming

Table of Contents

Hoofdstuk 1: Programmeertalen & Programmeren

Een programma bepaalt hoe we met invoer een bepaalde uitvoer kunnen verkrijgen.

[invoer uitvoer] | *images/invoer-uitvoer.gif*

Excel werkt met bepaalde invoer (bijvoorbeeld cijfergegevens) en kan deze verwerken naar een bepaalde uitvoer (bijvoorbeeld een grafiek). Excel is dus het programma dat de verwerking uitvoert. Gebruikers kunnen verschillende programma's gebruiken of installeren. Dit om met andere invoer te werken of de verwerking anders uit te voeren. Als programmeur kan je echter het programma maken dat de uitvoer doet.

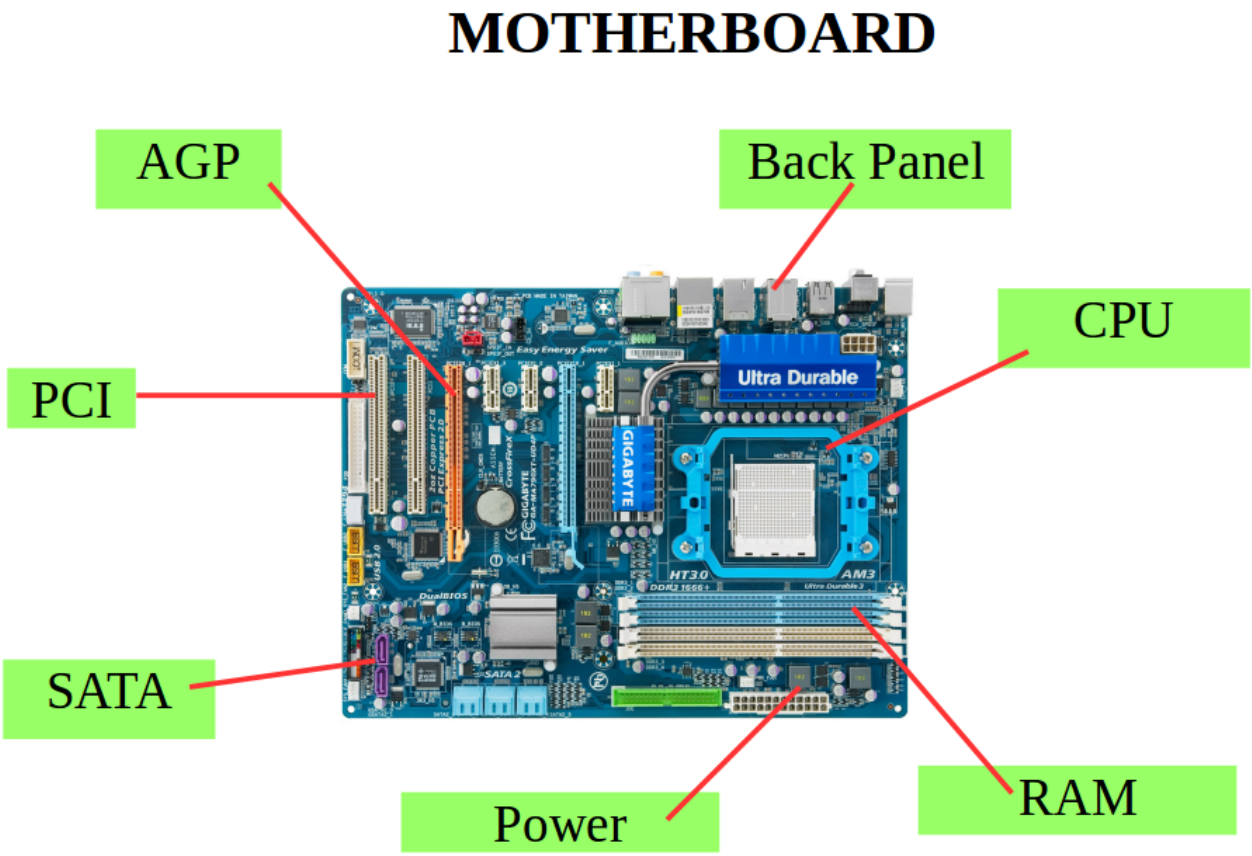
Opdrachten voor de computer

Een computer is een logische machine die we opdrachten kunnen geven.

Voorbeelden van opdrachten:

- Schrijf iets weg naar een bestand op de ssd.
- Krijg een lijst van de bestanden op de hdd.
- Toon iets op het scherm.
- Haal een pagina op van het internet via het ethernet netwerk.
- Tel 2 getallen op en sla het op in een variabele x.

Opdrachten worden gegeven via een programma. De computer bestaat uit verschillende onderdelen. Het is de CPU, processor of centrale rekeneenheid die het programma interpreteert.

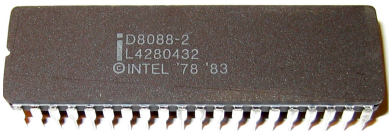


Opdrachten geven aan een computer kan via een programma. Met een programmeertaal kan een programma geschreven worden.

CPU Instructieset

Iedere computer **CPU** heeft een bepaalde **architectuur** of ontwerp. Hier zijn heel wat standaarden in:

- x86 (32bit), ontwikkeld door Intel

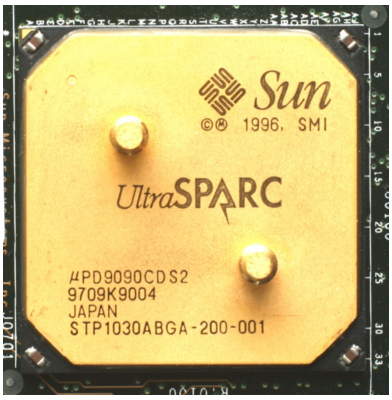


- x64 (64bit), de modernere variant ontwikkeld door AMD

- ARM, een zuinigere architectuur, deze zit bijna iedere smartphone.

  [ARM cpu] | *images/arm.webp*

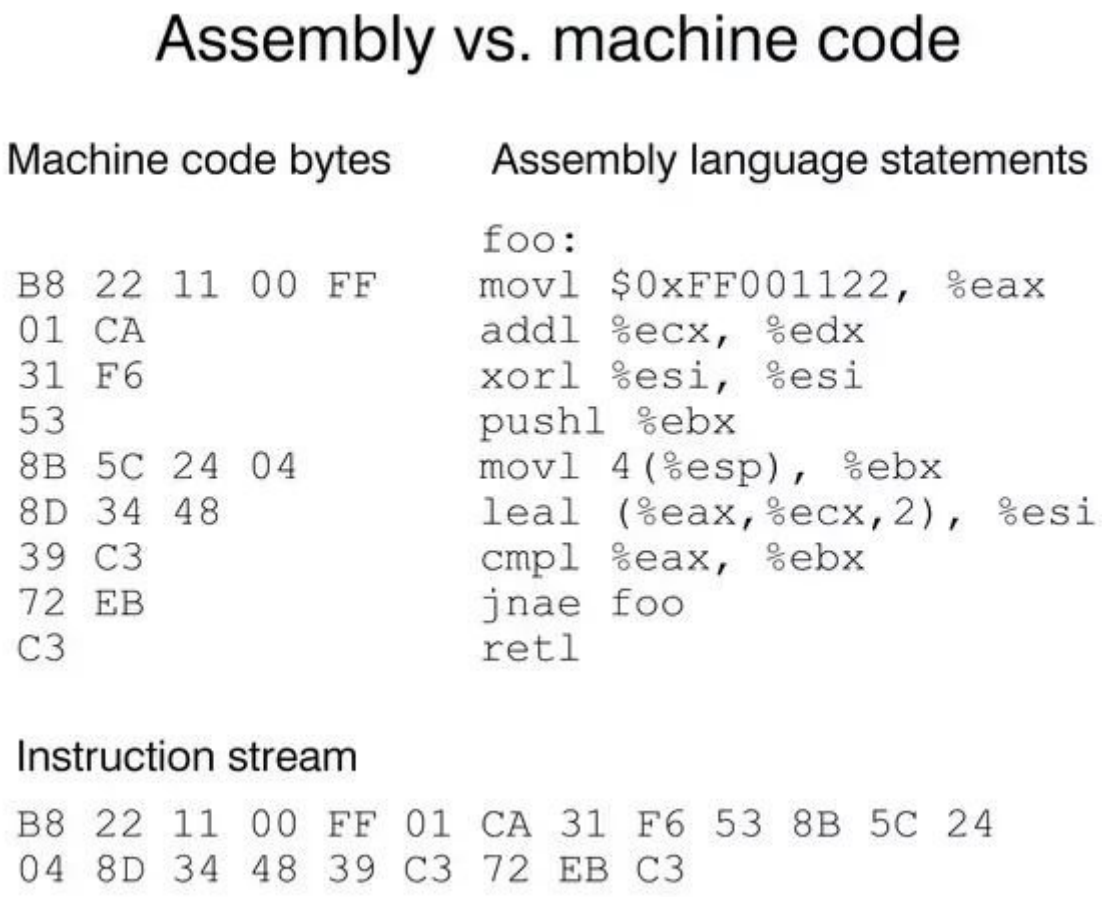
- RISC



- Mainframe

Aan deze architectuurstandaarden van CPU's kunnen opdrachten gegeven worden via **de instructieset**, dit is de machinetaal van de processor. Dit zijn afgesproken opdrachten op een laag niveau (vb. onthoud een getal, tel een getal op, haal een waarde op van de ssd, …)

Deze opdrachten via de instructieset kunnen in een laag niveau computertaal worden voorgesteld, namelijk *assembly*:



Dit is een **zeer ingewikkelde manier om te programmeren**.

Gelukkig zijn er hoog niveau programmeertalen ontwikkeld die het ons gemakkelijker maken.

Hoog niveau programmeertalen

Hoog niveau programmeertalen (die makkelijker zijn om te schrijven), kunnen via compilatie automatisch omgezet worden naar machinetaal.

Tekst (Hogere programmeertaal) → compilatie → machinetaal voor een specifieke architectuur

De machinetaal wordt bijgehouden in een uitvoerbaar bestand of *executable* (iets waarop je kan dubbelklikken om het programma te openen).

Voorbeelden *Hello World* in enkele programmeertalen

- C

```c
#include <stdio.h>

int main()
{
    printf("Hello World");
    return 0;
}
```

- Cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World";
    return 0;
}
```

- Ruby

```ruby
puts "Hello World!"
```

- Java

```java
    class MyClass{
        public static void main(String args[]){
         System.out.println("Hello Java");
        }
    }
```

- Python 3

```
print("Hello, World!")
```

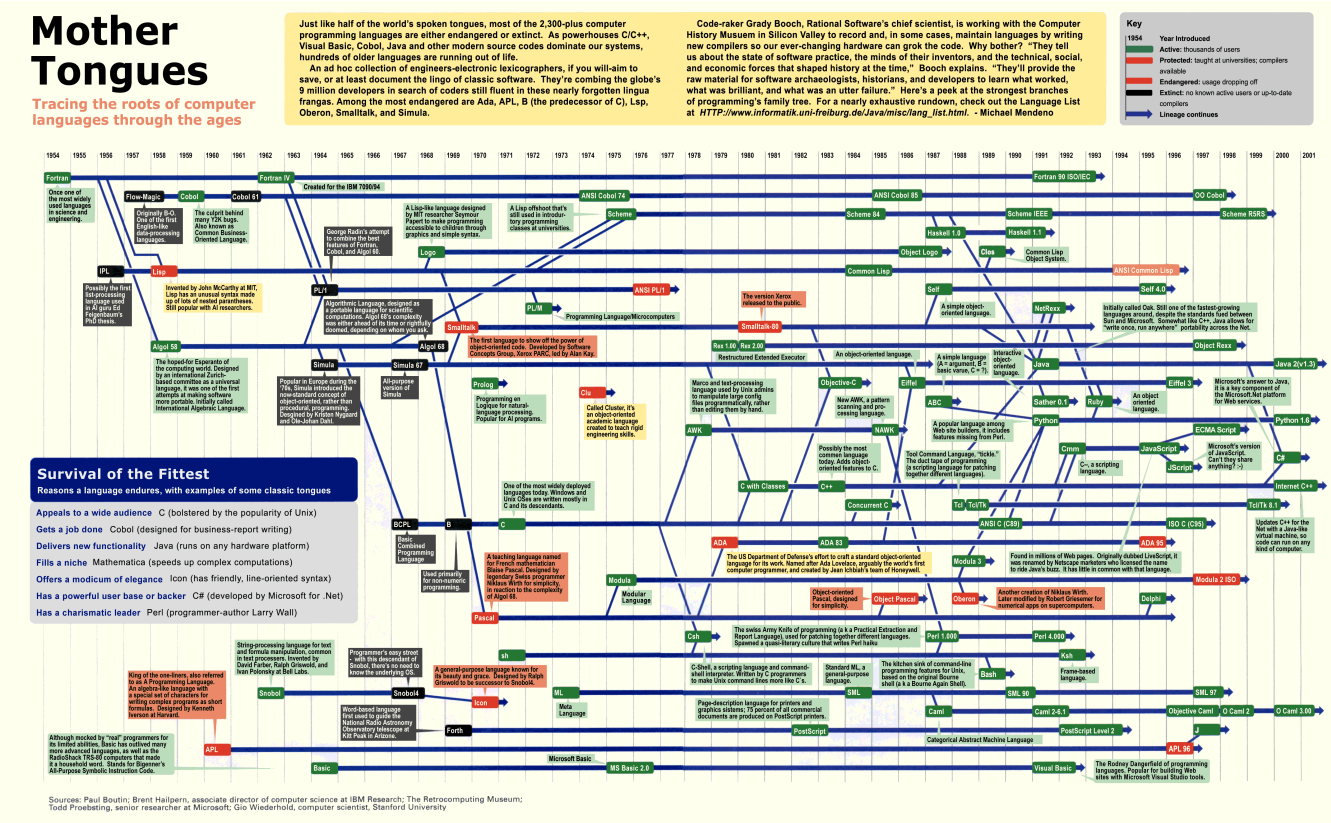
- C#

```
using System;
namespace HelloWorldApp {
    class MyClass {
        static void Main(string[] args) {
            Console.WriteLine("Hello World!");
            Console.ReadKey();
        }
    }
}
```

- Fortran

```
program hello
  print *, 'Hello, World!'
end program hello
```

Programmeertalen: Een historisch overzicht



Er zijn voortdurend nieuwe ontwikkelingen in programmeertalen. Nieuwe talen worden ontwikkeld. Ze kunnen populair worden, bijna niet gebruikt worden of andere talen beïnvloeden

met hun nieuwe ideeën. Iedere bestaande taal, zoals Python, wordt ook verder doorontwikkeld. Het verschil tussen python versie 1 en versie 3.11 is enorm groot. Hoewel het om dezelfde programmeertaal gaat, zal een programma ontwikkeld voor pyhton 3, niet meer werken op python 1.

De Geschiedenis van Programmeertalen

De geschiedenis van programmeertalen is een evolutie gedreven door innovatie om aan steeds veranderende softwarebehoeften te voldoen. Laten we enkele belangrijke mijlpalen in de geschiedenis van programmeertalen nader bekijken:

- **1940s-1950s:** De allereerste programmeertalen, zoals Assembly en Fortran, waren voornamelijk gericht op machinetaal en technische details.
- **1950s-1960s:** Talen zoals COBOL en ALGOL introduceerden hogere abstractie en leesbaarheid, wat programmeren toegankelijker maakte.
- **1960s-1970s:** De opkomst van talen zoals C en Pascal bracht gestructureerd programmeren en modulariteit naar voren.
- **1980s-1990s:** Objectgeoriënteerde talen zoals C++ en Java versterkten concepten zoals herbruikbaarheid en complexe softwareontwikkeling.
- **2000s-heden:** Moderne talen zoals Python leggen de nadruk op leesbaarheid, productiviteit en veelzijdigheid, waardoor ze geschikt zijn voor diverse toepassingen.

De programmeertaal Python

In deze cursus programmeren werken we met de programmeertaal Python. Hier zijn verschillende redenen voor:

- Python is **open source**, wat betekent dat je het gratis kunt gebruiken zonder licentiekosten.
- Python is een **krachtige programmeertaal**.
- Het is een **objectgeoriënteerde** taal en werkt met klassen naast functies.
- Python is een **multiplatform** programmeertaal. Zodra de code is geschreven, werkt deze op Windows, Linux, MacOSX, en meer.
- Python heeft een functionele programmeerstijl, naast het objectgeoriënteerde aspect (waardoor het een meervoudige paradigma-taal is).
- Python is **populair** in het bedrijfsleven en in diverse sectoren.
- Python maakt gebruik van veel concepten die ook in andere programmeertalen voorkomen. Als je eenmaal Python beheerst, kun je gemakkelijk overstappen naar andere talen.

Hoewel Python over het algemeen minder snel is dan Java vanwege zijn dynamische aard, heeft het voortdurende optimalisaties en externe modules die de prestaties kunnen verbeteren, wat de snelheidskloof in veel gevallen verkleint.

Meer informatie kan je hier terugvinden (dit behoort niet tot de te kennen leerstof).

Python schrijven (of programmeren)

Python wordt geschreven in een tekst document. Dit tekst document heeft de .py extensie. Bijvoorbeeld: MijnPythonBestand.py . Dit tekstbestand bevat gewone tekst (woorden bestaande uit karakters). De tekst die je schrijft moet wel voldoen aan de syntax van de taal.

De syntax

Iedere programmeertaal bestaat uit een syntax. Dit zijn een reeks afspraken van de taal:

1. **Welke woorden** kunnen gebruikt worden? In Python mag een woord bijvoorbeeld niet beginnen met een cijfer.
2. welke woorden zijn **gereserveerd** door de programmeertaal? Bijvoorbeeld het *return* woord.
3. Op welke **plaats** mogen deze woorden staan?
4. Welke **scheidingskarakters** worden toegestaan of verplicht? Python verplicht bijvoorbeeld *:* na een for lus.
5. Hoe belangrijk is **indentatie**?

Om succesvol te programmeren dien je de syntax van een taal te kennen.

De syntax bepaalt hoe statements, variabelen, functies en klassen geschreven moeten worden.

Waarom Python kiezen in plaats van Java?

Bij het kiezen van een programmeertaal, zoals Python of Java, rijst vaak de vraag welke het meest geschikt is. Hoewel beide talen krachtige tools zijn voor softwareontwikkeling, zijn er enkele overwegingen die Python tot een aantrekkelijke keuze maken, vooral voor beginners en middelbare scholieren.

Leesbaarheid en Eenvoud

Python onderscheidt zich door zijn eenvoudige en leesbare syntaxis. Dit betekent dat code in Python bijna als natuurlijke taal lijkt en gemakkelijk te begrijpen is, zelfs voor mensen zonder programmeerachtergrond. Laten we dit vergelijken met een eenvoudig voorbeeld in zowel Python als Java:

```
# Python
print("Hello, world!")

# Java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Zoals je kunt zien, is de Python-code veel beknopter en leesbaarder. Dit maakt het eenvoudiger om de essentie van de code te begrijpen zonder afgeleid te worden door overmatige syntactische details.

Snel Prototypen en Snelle Ontwikkeling

Python staat bekend om zijn snelle ontwikkelingsmogelijkheden. Het vereist minder code om dezelfde functionaliteit te bereiken in vergelijking met talen zoals Java. Dit is vooral handig bij het maken van prototypes en het iteratief ontwikkelen van projecten. Laten we een vergelijking maken tussen een eenvoudige lijstverwerkingstaak in beide talen:

```
# Python
numbers = [1, 2, 3, 4, 5]
squared = [num ** 2 for num in numbers]

# Java
import java.util.ArrayList;

public class SquaredNumbers {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        numbers.add(5);

        ArrayList<Integer> squared = new ArrayList<>();
        for (Integer num : numbers) {
            squared.add(num * num);
        }
    }
}
```

In dit voorbeeld toont Python zijn beknopte syntaxis en vermogen om complexe operaties te vereenvoudigen. Dit kan leiden tot een productievere ontwikkeling en snellere resultaten, vooral voor beginners.

Laten we nog een eenvoudig voorbeeld bekijken om het verschil tussen Python en Java te illustreren bij het uitvoeren van een eenvoudige berekening, bijvoorbeeld het berekenen van de som van getallen van 1 tot 10.

```
# Python
total = sum(range(1, 11))
print("De som is:", total)
```

```java
// Java
public class SumExample {
    public static void main(String[] args) {
        int total = 0;
        for (int i = 1; i <= 10; i++) {
            total += i;
        }
        System.out.println("De som is: " + total);
    }
}
```

Zoals je kunt zien, vereenvoudigt Python de berekening en resulteert in minder code.

Brede Toepassingsmogelijkheden

Hoewel Java een krachtige taal is voor het bouwen van complexe systemen, biedt Python een breed scala aan toepassingen en domeinen. Python wordt gebruikt voor webontwikkeling, gegevensanalyse, machine learning, wetenschappelijke berekeningen en nog veel meer. Het is zelfs een populaire taal voor scripting en automatisering. Deze veelzijdigheid stelt ontwikkelaars in staat om hun vaardigheden over verschillende disciplines toe te passen zonder van taal te hoeven wisselen.

- **Webontwikkeling:** Met frameworks zoals Django en Flask kun je interactieve en dynamische websites bouwen.
- **Data-analyse:** Python wordt vaak gebruikt in combinatie met bibliotheken zoals pandas en NumPy om gegevens te analyseren en inzichten te verkrijgen.
- **Wetenschappelijke berekeningen:** Bibliotheken zoals SciPy en Matplotlib stellen wetenschappers in staat om complexe berekeningen uit te voeren en resultaten te visualiseren.
- **Machine learning en kunstmatige intelligentie:** Populaire bibliotheken zoals TensorFlow en scikit-learn maken geavanceerde AI-implementaties mogelijk.
- **Automatisering:** Python kan worden gebruikt om repetitieve taken te automatiseren en workflows te stroomlijnen.

Conclusie

Hoewel Java en Python beide waardevolle programmeertalen zijn, biedt Python enkele voordelen die het aantrekkelijk maken voor beginners en middelbare scholieren. De leesbaarheid, eenvoudige syntaxis, snelle ontwikkeling en brede toepassingsmogelijkheden maken Python een uitstekende keuze om te leren en te beginnen met programmeren. In deze cursus zullen we ons richten op Python vanwege zijn geschiktheid voor beginners en de kansen die het biedt voor het opbouwen van solide programmeervaardigheden.

Hoofdstuk 2: Van Code naar Uitvoering op de CPU

Deze python code print een tekst naar de console:

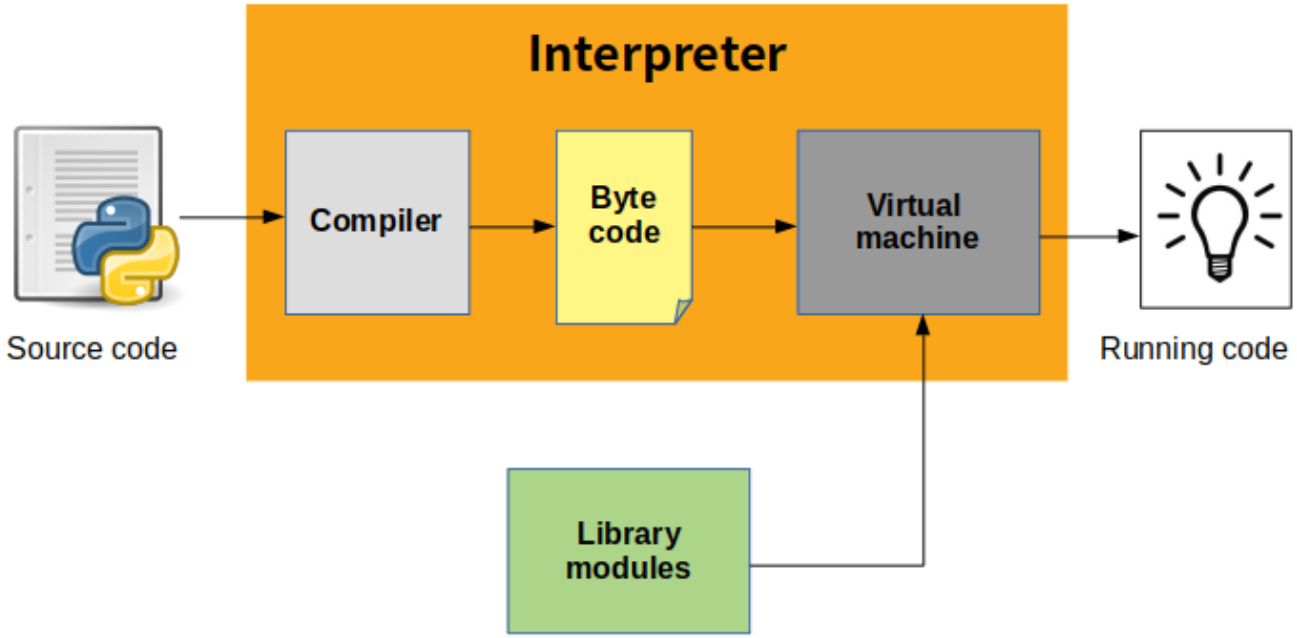
```
print('Hello world!')
```

Maar toch wordt deze codeopdracht niet rechtstreeks uitgevoerd op de CPU. De CPU werkt immers enkel op machinecode instructies.

Er worden dus enkele tussenliggende stappen uitgevoerd.

De Rol van de Interpreter

Wanneer je Python-code schrijft, schrijf je in een taal die begrijpelijk is voor mensen, maar niet direct voor computers. Om deze code uit te voeren, wordt een cruciale tussenstap ingezet: de Python-interpreter.



De Python-interpreter vertaalt de menselijk leesbare Python-code naar een vorm van tussentaal die bekend staat als bytecode. Bytecode is een reeks instructies die specifiek zijn ontworpen om te worden uitgevoerd door de Python Virtual Machine (PVM) Dit virtuele uitvoeringssysteem fungeert als een brug tussen de abstracte code en de werkelijke uitvoering op de CPU.

[python virtual machine] | *images/python-virtual-machine.webp*

Dit is bijvoorbeeld de gegenereerde bytecode van het 'Hello world' programma:

```
\00\00\00\00\EA\D5\E5d\00\00\00\E3\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00
\00\00@\00\00\00s\00\00\00e\00d\00\83\00d\00\00)\zHello
world!N)\DA print\A9\00r\00\00\00r\00\00\00\FAR/home/mark/Documents/python/liclipse-
workspace/pythontest/pythoncode/helloworld.py\DA<module>\00\00\00s\00\00\00\00\00
```

Voor ons is dit moeilijk leesbaar, maar de Python virtual machine kan dit perfect lezen.

Van Bytecode naar Machinecode

Machinecode is de laagste niveau van instructies die de CPU direct begrijpt. Echter, voordat de machinecode wordt bereikt, wordt de bytecode verwerkt door de PVM, die het omzet in instructies die kunnen worden uitgevoerd door de fysieke CPU van de computer.

De PVM is in feite een interpreter voor bytecode die ervoor zorgt dat de code wordt uitgevoerd op verschillende platforms (Windows, Linux, MacOSX, … , maar ook de processor op dit systeem: x86, ARM, x64, RISC,..) zonder dat het nodig is om dezelfde code te herschrijven voor elk platform. Dit maakt Python een platformonafhankelijke taal.

Linter: Codekwaliteit en Conventies

Voordat de code wordt uitgevoerd, is het van groot belang om ervoor te zorgen dat deze correct is en voldoet aan specifieke coderingsstandaarden Een linter is een hulpmiddel dat de code analyseert op fouten, inconsistenties en afwijkingen van de conventies.

Laten we dit illustreren met een voorbeeld van een stukje Python-code:

```python
# Onjuiste naamgeving van variabelen
Var = 42
prinT(var)
```

Een linter zou hier waarschuwingen genereren voor onjuiste naamgeving van variabelen en een functie die niet correct is gespeld.

Compilatie en Uitvoering

Python wordt beschouwd als een geïnterpreteerde taal, wat betekent dat de code regel voor regel wordt uitgevoerd door de interpreter. Dit in tegenstelling tot talen zoals C++ of Java, waarbij de code eerst volledig wordt omgezet in machinecode door een **compilatieproces** voordat deze wordt uitgevoerd.

Het Interpreterproces van de interpreter

[lexer] | *images/lexer.webp*

1. Tokenizing (lexer): De broncode wordt **opgesplitst in elementaire eenheden**, tokens genaamd. Deze tokens omvatten sleutelwoorden, symbolen, variabelen en getallen.

2. Parsing: De tokens worden **geanalyseerd** om de **syntactische structuur** van het programma te begrijpen. Hierbij wordt gecontroleerd of de code voldoet aan de grammaticaregels van Python.
3. Vertalen: De geparste code wordt omgezet in **tussentijdse bytecode**. Dit is een binair formaat dat de computer begrijpt en dat sneller kan worden uitgevoerd dan de originele broncode.
4. Uitvoering: De bytecode wordt uitgevoerd door de **Python-runtime**. Tijdens deze stap worden variabelen gemaakt, waarden toegewezen, bewerkingen uitgevoerd en resultaten gegenereerd.

Deze 4 stappen zijn een voortdurende lus, die bij iedere regel (dus regel per regel) wordt uitgevoerd.

Een voorbeeld

Stel je voor dat je een Python-script hebt dat de beweging van een speler in een game simuleert:

```
initial_position = 0
movement_speed = 5

new_position = initial_position + movement_speed
print("Nieuwe positie van de speler:", new_position)
```

Bij het uitvoeren van dit script zal de interpreter de variabelen initial_position en movement_speed aanmaken en waarden toewijzen. Vervolgens wordt de positie van de speler berekend door movement_speed op te tellen bij initial_position. Het resultaat wordt afgedrukt.

Just-In-Time (JIT) Compiler in Python

De Just-In-Time (JIT) compiler is een techniek die wordt gebruikt om de uitvoeringssnelheid van code te verbeteren door delen van de code tijdens de uitvoering te compileren naar machinecode. Hoewel Python een geïnterpreteerde taal is, kan de JIT-compiler de prestaties aanzienlijk verbeteren door bepaalde stukken code te compileren op het moment dat ze worden uitgevoerd.

Hoe Werkt de JIT-Compiler?

Normaal gesproken wordt Python-code geïnterpreteerd en vertaald naar tussentijdse bytecode. Bij gebruik van een JIT-compiler worden sommige delen van de code, die vaak worden uitgevoerd, tijdens de uitvoering gecompileerd naar directe machinecode. Dit maakt de uitvoering sneller omdat machinecode direct door de computer kan worden uitgevoerd zonder interpretatie.

Voorbeeld

Stel je voor dat je een eenvoudig Python-script hebt voor het simuleren van de beweging van een speler in een game:

```python
def simulate_movement(initial_position, movement_speed):
    new_position = initial_position
    for _ in range(1000000):
        new_position += movement_speed
    return new_position

initial_position = 0
movement_speed = 5

final_position = simulate_movement(initial_position, movement_speed)
print("Laatste positie van de speler:", final_position)
```

In dit voorbeeld wordt de functie simulate_movement herhaaldelijk (meerdere keren per seconde) opgeroepen om de beweging van een speler te simuleren. Python kan na een aantal uitvoeringen ervoor kiezen om de simulate_movement-functie te optimaliseren door het om te zetten in efficiënte machinecode (via de JIT compiler).

Dit proces gebeurt automatisch, hier heb je als programmeur geen invloed op.

Voordelen in Gaming

In de context van gaming kan de JIT-compiler de prestaties van Python-code verbeteren, vooral voor onderdelen van het spel die veelvuldig worden gebruikt, zoals fysica-simulaties, beeldverwerking of AI-berekeningen. Dit kan leiden tot soepelere en responsievere spelervaringen.

Hoewel JIT-compilatie Python niet zo snel maakt als strikt gecompileerde talen, kan het aanzienlijke prestatieverbeteringen bieden.

Voordelen en Nadelen

Python's interpretatieproces biedt voordelen zoals **directe feedback** en gemakkelijke debugging. Echter, omdat elke regel code wordt geïnterpreteerd wanneer deze wordt uitgevoerd, kan Python trager zijn dan gecompileerde talen voor intensieve berekeningen in games.

Ondanks enige **snelheidsbeperkingen** blijft Python een veelgebruikte taal in de game-ontwikkeling vanwege zijn flexibiliteit, eenvoudige syntax en uitgebreide bibliotheken.

Samenvatting

Het pad van Python-code naar uitvoering op de CPU omvat diverse tussenstappen, van interpretatie en bytecode tot JIT-compilatie en uiteindelijk machinecode.

Hoofdstuk 3: Gebruik van Variabelen

Variabelen in Python

Variabelen vormen een kernconcept in programmeren, waardoor softwareontwikkelaars waarden

kunnen opslaan en manipuleren voor gebruik in hun programma's. In Python zijn variabelen containers voor het opslaan van gegevens zoals getallen, tekst, lijsten, en meer.

Variabelen Toewijzen

Variabelen in Python worden gemaakt door een naam toe te wijzen aan een waarde. Hier is een voorbeeld waarin we de *health* van een speler in een game bijhouden:

python

```python
# Player health in Counter-Strike
player_health = 100
print("Huidige gezondheid:", player_health)
```

In dit voorbeeld wordt de variabele player_health aangemaakt en toegewezen met de waarde 100. We kunnen de naam van de variabele gebruiken om later de opgeslagen waarde te manipuleren.

Naamgeving van Variabelen

In Python is het benoemen van variabelen van groot belang om duidelijke en begrijpelijke code te schrijven. Door een consistente en zinvolle naamgeving te hanteren, maak je niet alleen je code leesbaarder, maar vergemakkelijk je ook het onderhoud en de samenwerking aan projecten. Hier zijn enkele belangrijke regels en tips voor het benoemen van variabelen in Python:

Regels voor Naamgeving van Variabelen in Python

- Geldigheid: Variabelen moeten met een letter (a-z, A-Z) of een underscore (_) beginnen. Ze kunnen ook cijfers (0-9) bevatten, maar niet als eerste teken. Bijvoorbeeld: player_score, _health, level1.
- Hoofdlettergevoeligheid: Python is hoofdlettergevoelig, wat betekent dat variabelen met verschillende hoofdletters als verschillende namen worden beschouwd. score en Score zouden bijvoorbeeld twee verschillende variabelen zijn.
- Spaties en Speciale Tekens: Spaties en speciale tekens zoals @, $, en % zijn niet toegestaan in variabelennamen.
- Spaties Vermijden: In plaats van spaties worden underscores vaak gebruikt om woorden in variabelennamen te scheiden, wat de leesbaarheid verbetert. Bijvoorbeeld: player_name, health_points.
- Betekenisvolle Namen: Geef je variabelen namen die hun functie of betekenis in de context aangeven. Een naam als player_health is duidelijker dan een generieke naam zoals x.

Datatypen en Variabelen

In Python zijn variabelen **niet expliciet getypeerd**, wat betekent dat je geen datatype hoeft op te geven wanneer je een variabele declareert. Python bepaalt automatisch het datatype op basis van de waarde die aan de variabele wordt toegewezen.

Dit maakt van Python een dynamisch getypeerde taal.

Variabelen in Python worden toegewezen met behulp van de toewijzingsoperator (`=`). De operator wijst een waarde toe aan een variabele.

Bijvoorbeeld, om een integer (geheel getal) op te slaan in een variabele genaamd `bullet_damage`, hoef je alleen maar deze toewijzing uit te voeren:

```
bullet_damage = 16
```

In dit geval heeft Python automatisch het datatype van `bullet_damage` ingesteld op een integer met waarde 16.

In bijvoorbeeld Java, moet je het gegevenstype wel expliciet meegegeven:

```
int bullet_damage = 16
```

Waarden Bijwerken

Variabelen kunnen op elk moment worden bijgewerkt door er simpelweg een nieuwe waarde aan toe te wijzen. Bijvoorbeeld, als de speler schade oploopt:

```
# Schade oplopen
damage_taken = 25
player_health -= damage_taken
print("Nieuwe gezondheid na geraakt te zijn:", player_health)
```

Hier wordt de variabele damage_taken gemaakt en toegewezen met de waarde 25. Vervolgens verminderen we de gezondheid van de speler door damage_taken van player_health af te trekken.

Variabelen en Scope

Variabelen hebben een bepaalde "scope" of bereik, wat aangeeft waar de variabele beschikbaar en geldig is. In Python kunnen variabelen lokaal (binnen een specifieke functie) of globaal (door het hele programma) zijn.

Laten we een voorbeeld bekijken om het concept van scope te illustreren:

```
globale_variabele = 10

# definitie van een functie
def functie():
    lokale_variabele = 5
    print("Lokale variabele, binnen functie:", lokale_variabele)
    print("Globale variabele, binnen functie:", globale_variabele)

# nu voeren we de functie uit
functie()

# En we voeren enkele print statements uit
print("Globale variabele, buiten functie:", globale_variabele)
print("Lokale variabele, buiten functie:", lokale_variabele)
```

Uitvoer:

```
Lokale variabele, binnen functie: 5 ①
Globale variabele, binnen functie: 10 ②
Globale variabele, buiten functie: 10 ③
Traceback (most recent call last): ④
  File "test.py", line 14, in <module>
    print("Lokale variabele, buiten functie:", lokale_variabele)
NameError: name 'lokale_variabele' is not defined. Did you mean: 'globale_variabele'?
⑤
```

① De python interpreter zit in de functie, lokale variabele gevonden.

② De python interpreter zit in de functie, globale variabele gevonden.

③ De python interpreter zit buiten de functie, globale variabele gevonden.

④ De python interpreter zit buiten de functie, lokale variabele NIET gevonden.

⑤ Python probeert zelfs een juiste oplossing voor te stellen. Het blijft aan de ontwikkelaar om iets met deze suggestie te doen.

Hier zien we dat de variabele `lokale_variabele` alleen beschikbaar is binnen de functie waarin deze is gedefinieerd. De variabele `globale_variabele` kan zowel binnen als buiten de functie worden gebruikt, omdat deze een globale scope heeft.

Variabelen en Gegevenstypen

Python heeft verschillende ingebouwde gegevenstypes die je kunt gebruiken om variabelen van verschillende soorten gegevens op te slaan. Enkele van de veelgebruikte datatypen zijn:

- **int:** Gehele getallen, zoals 5, -10, 100.
- **float:** Komma-getallen, zoals 3.14, -0.5, 2.0.
- **str:** Tekst, zoals "Hallo, wereld!", 'Python'.

- **bool:** Booleaanse waarden, True of False.

Bijvoorbeeld:

```
a = 5          # int
b = 3.14       # float
naam = "Alice" # str
waar = True    # bool
```

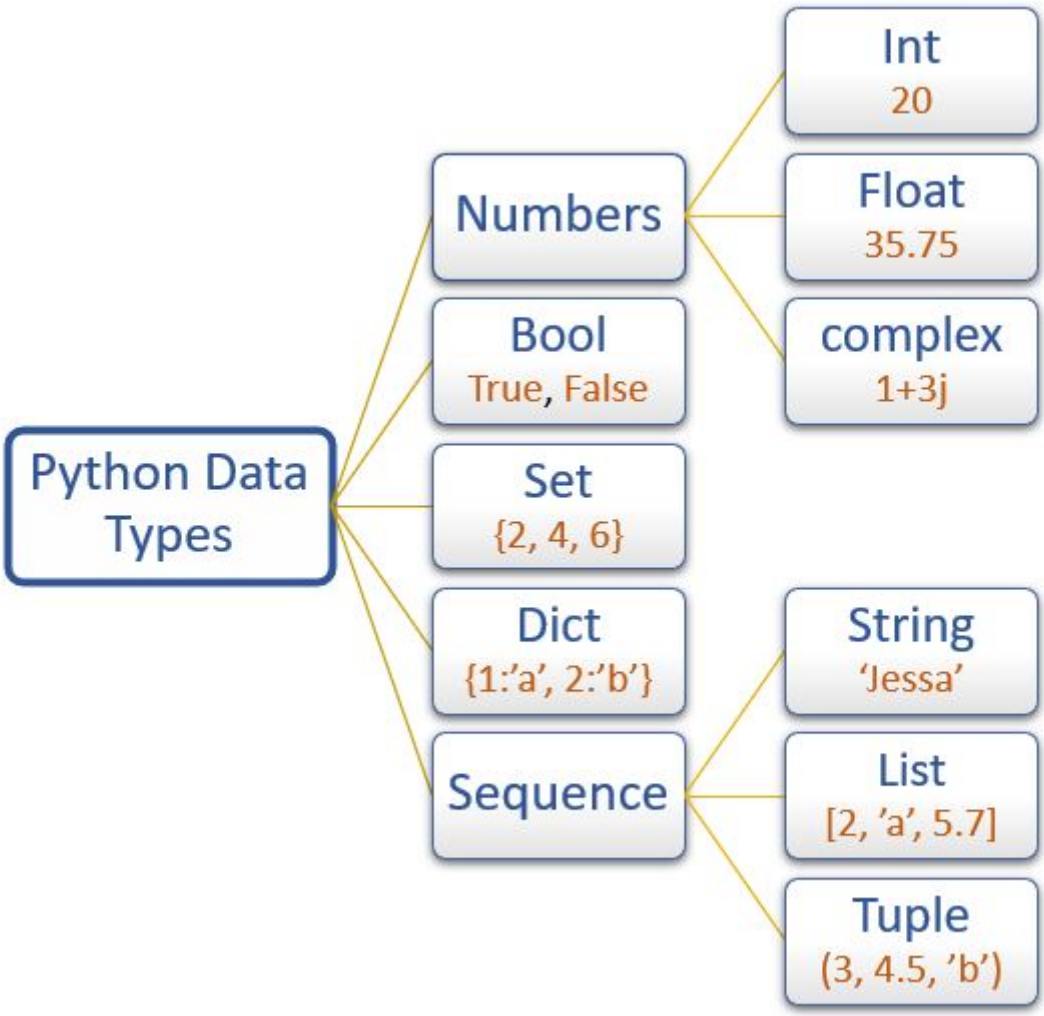


Figure 1. De standaard Python gegevenstypen

Gegevenstypes bepalen hoe de computer gegevens opslaat, bewerkt en weergeeft. In een gamingcontext kunnen gegevenstypes worden gebruikt om informatie zoals spelergezondheid, scores, positie en meer vast te leggen.

Laten we de belangrijkste gegevenstypes eens overlopen:

Integer (int)

Het gegevenstype `int` staat voor gehele getallen, zoals 5, -10 en 100. Integer-waarden worden gebruikt voor wiskundige bewerkingen en numerieke berekeningen. Python staat toe om positieve en negatieve gehele getallen te gebruiken, evenals getallen met en zonder decimale punten.

Bijvoorbeeld:

```
leeftijd = 16
temperatuur = -5
aantal_studenten = 30
```

Float (Komma-getal)

Het gegevenstype `float` vertegenwoordigt komma-getallen, ook wel bekend als zwevendekomma-getallen. Dit omvat getallen met decimale punten, zoals 3.14, -0.5 en 2.0. Float-waarden worden gebruikt voor nauwkeurige berekeningen met reële getallen.

Bijvoorbeeld:

```
pi = 3.14159
geldbedrag = 123.45
hoogte = -10.5
```

String (str)

Het gegevenstype `str` staat voor tekstuele gegevens, zoals woorden, zinnen of karakters. Tekst in Python wordt omringd door enkele aanhalingstekens (') of dubbele aanhalingstekens (").

Bijvoorbeeld:

```
naam = "Alice"
bericht = 'Hallo, wereld!'
label = "Productcode: 12345"
```

Strings kunnen worden gecombineerd (geconcateneerd) met behulp van de `+` operator:

```
voornaam = "John"
achternaam = "Doe"
volledige_naam = voornaam + " " + achternaam
print(volledige_naam) # Output: John Doe
```

Boolean (bool)

Het gegevenstype `bool` vertegenwoordigt booleaanse waarden, namelijk `True` (waar) of `False` (onwaar). Booleaanse waarden worden veel gebruikt in logische bewerkingen en beslissingsstructuren, zoals `if`-voorwaarden.

Bijvoorbeeld:

```
is_student = True
heeft_toegang = False
is_regenachtig = True
```

Booleaanse waarden zijn ook het resultaat van vergelijkingsoperatoren, zoals == (gelijk aan), != (niet gelijk aan), < (kleiner dan), > (groter dan), etc.

Stel je voor dat we een script hebben dat controleert of een speler voldoende gezondheid heeft om een bepaalde taak uit te voeren:

```
player_health = 75
minimum_health_required = 50

is_healthy_enough = player_health > minimum_health_required
print("Is de speler gezond genoeg?", is_healthy_enough)
```

In dit voorbeeld vergelijken we de gezondheid van de speler (player_health) met de vereiste minimale gezondheid (minimum_health_required). Als de gezondheid van de speler groter is dan het minimum, zal is_healthy_enough de waarde True krijgen. Anders zal het de waarde False krijgen.

Booleaanse waarden zijn krachtige hulpmiddelen in games omdat ze de besluitvorming en logica binnen je code mogelijk maken. Je kunt ze gebruiken om te bepalen of een speler een bepaald level heeft bereikt, of een missie is voltooid, of dat bepaalde acties kunnen worden uitgevoerd op basis van bepaalde voorwaarden.

Lijst Gegevenstype

list: Een geordende verzameling van waarden, zoals de inventaris van een speler.

```
inventory = ["sword", "shield", "potion"]
```

Tuple Gegevenstype

tuple: Een onveranderlijke verzameling van waarden, zoals de positie van een speler.

```
player_position = (10, 5)
```

Dictionary Gegevenstype

dict: Een verzameling van sleutel-waardeparen, zoals attributen van een speler.

```
player_attributes = {"name": "John", "health": 100, "armor": "plate"}
```

Type Conversie

Soms is het nodig om gegevens van het ene type naar het andere te converteren. Python biedt functies om dit te doen. Bijvoorbeeld, om een getal om te zetten naar een string, kun je de functie `str()` gebruiken:

```
leeftijd = 16
leeftijd_als_string = str(leeftijd)
```

Dynamische Typing

Python staat ook bekend om 'dynamic typed' te zijn, wat betekent dat het datatype van een variabele kan veranderen terwijl het programma wordt uitgevoerd Dit in tegenstelling tot sterk getypeerde talen waar het datatype strikt moet worden gedefinieerd en behouden.

Bijvoorbeeld:

```
a = 5
a = "Hallo"
a = True
```

Dit houdt een groot risico voor *bugs* in.

Variabelen vs Expressies

Een expressie is een combinatie van waarden, operatoren en/of functies die een resultaat oplevert wanneer deze wordt geëvalueerd. Expressies kunnen variëren van eenvoudige wiskundige bewerkingen tot complexere berekeningen. Ze kunnen ook variabelen bevatten.

```
damage = 25
health = 100
remaining_health = health - damage
```

In dit voorbeeld is health - damage een expressie die wordt geëvalueerd tot het resterende gezondheidsniveau van een speler na het oplopen van schade.

Een variabele is een naam die wordt gebruikt om een geheugenlocatie aan te duiden waarin gegevens kunnen worden opgeslagen. Variabelen worden gebruikt om waarden op te slaan en te bewaren, zodat ze later kunnen worden gebruikt in berekeningen, operaties en expressies.

```
player_health = 100
player_name = "Alice"
```

Hier worden player_health en player_name als variabelen gebruikt om respectievelijk de gezondheid van een speler en de naam van een speler op te slaan.

Het onderscheid tussen expressies en variabelen is dat expressies berekeningen uitvoeren en resultaten opleveren, terwijl variabelen dienen als namen voor opgeslagen gegevens Variabelen kunnen in expressies worden gebruikt om bewerkingen uit te voeren en nieuwe waarden te genereren.

Conclusie

De verscheidenheid aan gegevenstypes in Python, waaronder integer, float, string en boolean, biedt ontwikkelaars flexibiliteit bij het manipuleren van verschillende soorten gegevens. Het begrijpen van deze gegevenstypes en hun toepassingen is essentieel voor het schrijven van effectieve en veelzijdige Python-programma's.

Hoofdstuk 4: Beslissingsstructuren

Beslissingsstructuren stellen ons in staat om de logica van ons programma of spel aan te passen op basis van voorwaarden en situaties.

Het Belang van Beslissingsstructuren

Beslissingsstructuren zijn van cruciaal belang in het creëren van interactieve games. Ze stellen ons in staat om verschillende paden in het spel te volgen op basis van specifieke voorwaarden. Hierdoor kunnen we scenario's creëren waarin de uitkomst afhangt van de beslissingen van de speler.

Keuzes in een Tekstavontuur:

Stel je voor dat je een tekstgebaseerde RPG ontwikkelt waarin spelers een verlaten kasteel verkennen. Beslissingsstructuren kunnen worden gebruikt om verschillende resultaten te genereren op basis van keuzes:

```
print("Je staat voor een deur. Wat wil je doen?")
choice = "1"  # Stel dat de speler "1" invoert

if choice == "1":
    print("Je opent de deur en vindt een oude sleutel.")
else:
    print("Je doorzoekt de kamer verder maar vindt niets van belang.")
```

If-Else Beslissingsstructuur

Python biedt de `if-else`-structuur om beslissingen te nemen op basis van één enkele voorwaarde. Als de voorwaarde waar is, wordt de code binnen het `if`-blok uitgevoerd. Anders wordt de code binnen het `else`-blok uitgevoerd.

Beslissen of een Vijand Verslagen is:

Laten we een gevechtsscenario overwegen waarin de speler een vijand moet verslaan. Met een `if-else`-structuur kunnen we bepalen of de vijand verslagen is:

```
enemy_health = 30
player_damage = 25

if player_damage >= enemy_health:
    print("Je hebt de vijand verslagen!")
else:
    print("De vijand heeft nog", enemy_health - player_damage, "gezondheid over.")
```

If-Elif-Else Structuur

In complexere situaties waarin meerdere voorwaarden moeten worden geëvalueerd, biedt de `if-elif-else`-structuur uitkomst. Deze structuur stelt ons in staat om verschillende mogelijke uitkomsten te behandelen.

Beslissen over Spelerattributen:

Laten we een situatie aannemen waarin we de prestaties van een speler evalueren op basis van hun attributen:

```
player_health = 75
player_strength = 8
player_agility = 6

if player_health > 70:
    print("Je bent in goede conditie.")
elif player_health > 40:
    print("Je hebt wat herstel nodig.")
else:
    print("Je gezondheid is laag. Voorzichtig blijven!")
```

Geneste Beslissingsstructuren

Beslissingsstructuren kunnen ook genest zijn, wat betekent dat je een beslissing neemt binnen een ander beslissingsblok. Dit is nuttig om complexere logica te implementeren.

Keuzes in een Mysterieus Bos:

Stel je een game voor waarin spelers een mysterieus bos verkennen. Binnen dit bos kunnen ze verschillende paden kiezen, elk met hun eigen consequenties:

```
print("Je staat in een bos en ziet een splitsing.")
choice = "links"  # Stel dat de speler "links" invoert

if choice == "links":
    print("Je volgt het pad en vindt een verborgen schat!")
else:
    print("Je gaat een andere richting op en hoort vreemde geluiden.")
    second_choice = "doorgaan"  # Stel dat de speler "doorgaan" invoert
    if second_choice == "doorgaan":
        print("Je ontdekt een verlaten huis.")
    else:
        print("Je besluit om snel terug te keren en het enge gebied te verlaten.")
```

Het Belang van Indentatie in Python

Indentatie is een fundamenteel concept in Python-programmering en speelt een cruciale rol in het structureren van je code. In tegenstelling tot veel andere programmeertalen, waar accolades of sleutelwoorden worden gebruikt om codeblokken te definiëren, gebruikt Python indentatie om de structuur van het programma aan te geven. Dit kan van invloed zijn op leesbaarheid, logica en werking van je code.

Duidelijkheid en Leesbaarheid

Indentatie helpt om de structuur van je code visueel te benadrukken. Het geeft aan welke regels bij elkaar horen en vormt de basis van codeblokken. Een consistent gebruik van indentatie maakt het voor zowel jou als andere ontwikkelaars gemakkelijker om te begrijpen hoe de code is gestructureerd en welke delen met elkaar zijn verbonden.

Duidelijke Identatie:

```
if score > 100:
    print("Geweldig werk!")
    player_level += 1
    print("Je bent nu level", player_level)
```

Onjuiste Identatie:

```
if score > 100:
    print("Geweldig werk!")
print("Je bent nu level", player_level)
```

In het eerste voorbeeld wordt de tweede en derde regel uitgevoerd als de voorwaarde waar is. In het tweede voorbeeld wordt de derde regel altijd uitgevoerd, ongeacht de voorwaarde.

Logische Structurering

De juiste indentatie zorgt ervoor dat je code correct wordt uitgevoerd volgens de gewenste logica. Indentatie scheidt codeblokken, zoals loops, functies en conditionele statements, van elkaar. Het stelt Python in staat om de scope van variabelen en de uitvoering van instructies op de juiste manier te interpreteren.

Loop met Correcte Identatie:

```
for i in range(5):
    print(i)
    print("Dit is een iteratie van de loop.")
print("De loop is voltooid.")
```

Loop met Onjuiste Identatie:

```
for i in range(5):
    print(i)
print("Dit is een iteratie van de loop.")
print("De loop is voltooid.")
```

In het eerste voorbeeld worden de eerste twee regels binnen elke iteratie van de loop uitgevoerd. In het tweede voorbeeld worden deze regels na de voltooiing van de loop uitgevoerd, wat resulteert in een ander gedrag.

Foutpreventie

Foutieve indentatie kan leiden tot syntaxisfouten en logische fouten in je code. Python zal een foutmelding genereren als de indentatie niet correct is. Dit helpt je om snel fouten op te sporen en te corrigeren.

Indentatiefout:

```
if health > 0:
print("Je leeft nog!")
```

Python zal hier een foutmelding genereren omdat de code na de `if`-verklaring niet correct is ingesprongen.

Samenvatting

Indentatie is niet alleen een esthetisch aspect van Python-programmering, maar ook een fundamenteel onderdeel van hoe de taal werkt. Het zorgt voor leesbare, logische en foutvrije code. Door consequent de juiste indentatieregels te volgen, structureer je je code op een manier die gemakkelijk te begrijpen en te onderhouden is. Een goede indentatiepraktijk bevordert niet alleen

jouw codebase, maar ook samenwerking met andere ontwikkelaars en het creëren van betrouwbare software.

Hoofdstuk 5: Gegevensinvoer, Expressies en Berekeningen

In dit hoofdstuk gaan we in op gegevensinvoer en het gebruik van expressies en berekeningen.

Gegevensinvoer

Het vragen van gegevens van spelers of gebruikers is essentieel om je programma interactief te maken. Python biedt de input()-functie, die je in staat stelt om tekstuele invoer van spelers te verzamelen. Met deze functie kun je vragen stellen, keuzes aanbieden en spelers de mogelijkheid geven om direct met je game of programma te communiceren.

Tekstinvoer Verzamelen:

```
player_name = input("Voer je naam in: ")
print("Welkom, " + player_name + "!")
```

Met dit voorbeeld vragen we de speler om hun naam in te voeren en gebruiken we deze input om een persoonlijk welkomstbericht te genereren.

```
choice = input("Wil je naar links of rechts gaan? ")
print("Je hebt gekozen om naar " + choice + " te gaan.")
```

Hiermee kunnen spelers eenvoudige keuzes maken binnen je game en wordt de uitvoer aangepast op basis van hun invoer.

Expressies en Berekeningen

Het vermogen om expressies en berekeningen te gebruiken, opent een wereld van mogelijkheden in game-ontwikkeling. Hiermee kun je wiskundige en logische bewerkingen uitvoeren op basis van variabele waarden, invoer van spelers of gamegebeurtenissen.

Gevechtsberekening:

Stel je een RPG-game voor waarin spelers vijanden schade toebrengen. Hier is een voorbeeld van hoe je dit kunt aanpakken:

```
player_damage = int(input("Voer je schade in: "))
enemy_health = 100
remaining_health = enemy_health - player_damage

if remaining_health <= 0:
    print("De vijand is verslagen!")
else:
    print("De vijand heeft nog", remaining_health, "gezondheid over.")
```

Deze code gebruikt input() om de schade van de speler te vragen, en de berekening controleert of de vijand is verslagen of nog steeds in leven is.

Deze code gebruikt de invoer van de speler om de resterende gezondheid van een vijand na het toebrengen van schade te berekenen.

Attribuutberekening:

Stel je voor dat je een rollenspel maakt waarbij spelers de kracht en behendigheid van hun personage kunnen aanpassen:

```
player_name = input("Creëer je personage. Geef een naam: ")
player_strength = int(input("Voer de krachtwaarde in (1-10): "))
player_agility = int(input("Voer de behendigheidswaarde in (1-10): "))
total_points = player_strength + player_agility

print(player_name + "'s attributen:")
print("Kracht:", player_strength)
print("Behendigheid:", player_agility)
print("Totaal punten:", total_points)
```

Met dit voorbeeld verzamel je gegevens van spelers en gebruik je expressies om totaalpunten te berekenen op basis van kracht en behendigheid.

Verrijking van Gameplay via Interactie

Door de kracht van gegevensinvoer en berekeningen te combineren, kun je gameplay-ervaringen creëren die diep en meeslepend zijn. Stel je voor dat je een avonturenspel ontwikkelt waarin spelers een mysterie moeten oplossen:

```
player_name = input("Voer je naam in: ")
print("Welkom, " + player_name + "!")
print("Je staat voor een oud landhuis. Wat wil je onderzoeken?")
choice = input("1. De voordeur 2. Het zijraam: ")

if choice == "1":
    print("Je betreedt het landhuis via de voordeur en voelt een koude windvlaag.")
else:
    print("Je glijdt door het zijraam naar binnen en landt behendig op de vloer.")

print("Je onderzoek begint...")
```